Scala for Professionals

Object Functional Programming in Depth

Recursion

Recursion by Example: Factorial

For a natural number n, the factorial ($n!$) is defined recursively:

$$n! = \begin{cases} 1 & \text{if } n = 0, \\ (n-1)! \times n & \text{if } n > 0 \end{cases}$$

We can implement this in Scala in a basic recursive function:

```
def factorial(n: Int): BigInt =
  if (n == 0) 1 else n * factorial(n - 1)
```

- **Note**: Recursive functions in Scala *require* a type annotation.
- **Quiz**: What could go wrong with this implementation?

Stack Overflows, and Tail Recursion

Recursion can be stack intensive – depending on the stack size, `StackOverFlowErrors` may occur.

```
factorial(100000)
// java.lang.StackOverflowError
//   at .factorial(<console>:9)
```

- The Scala compiler can automatically optimize tail recursive operations, i.e. where the recursive call is the last instruction.
- **Best Practice**: When writing recursive methods, ensure they are tail recursive.

Compiler Verification of Tail Call Optimization

If we want to verify that an operation is tail recursive, use the `@tailrec` annotation:

```
import scala.annotation.tailrec

@tailrec
def factorial(n: Int): BigInt =
  if (n == 0) 1 else n * factorial(n - 1)
// <console>:14: error: could not optimize @tailrec annotated method
//   factorial: it contains a recursive call not in tail position
//         if (n == 0) 1 else n * factorial(n - 1)
//                             ^
```

By default, the Scala compiler applies tail call optimization silently to code that qualifies.*

---

* The JVM doesn't have tail call optimization – Scala performs a code transformation during compilation.

Tail Recursion By Example

We can introduce an accumulator in order to make our `factorial` method tail recursive:

```
@tailrec
def factorial(n: Int, acc: BigInt = 1): BigInt =
  if (n == 0) acc else factorial(n - 1, n * acc)
```

- **NOTE**: It isn't always this easy to make methods tail recursive.
- **Quiz**: Is there anything potentially wrong with this method?

Digression: Local Methods

Methods, just like variables, may be defined inside a method:

```
def length(x: Double, y: Double) = {
  def square(v: Double) = v * v
  sqrt(square(x) + square(y))
}

length(3, 4)
// res0: Double = 5.0
```

Improved Tail Recursion: By Example

The `acc` parameter in our tail recursive `factorial` could confuse callers of the method.

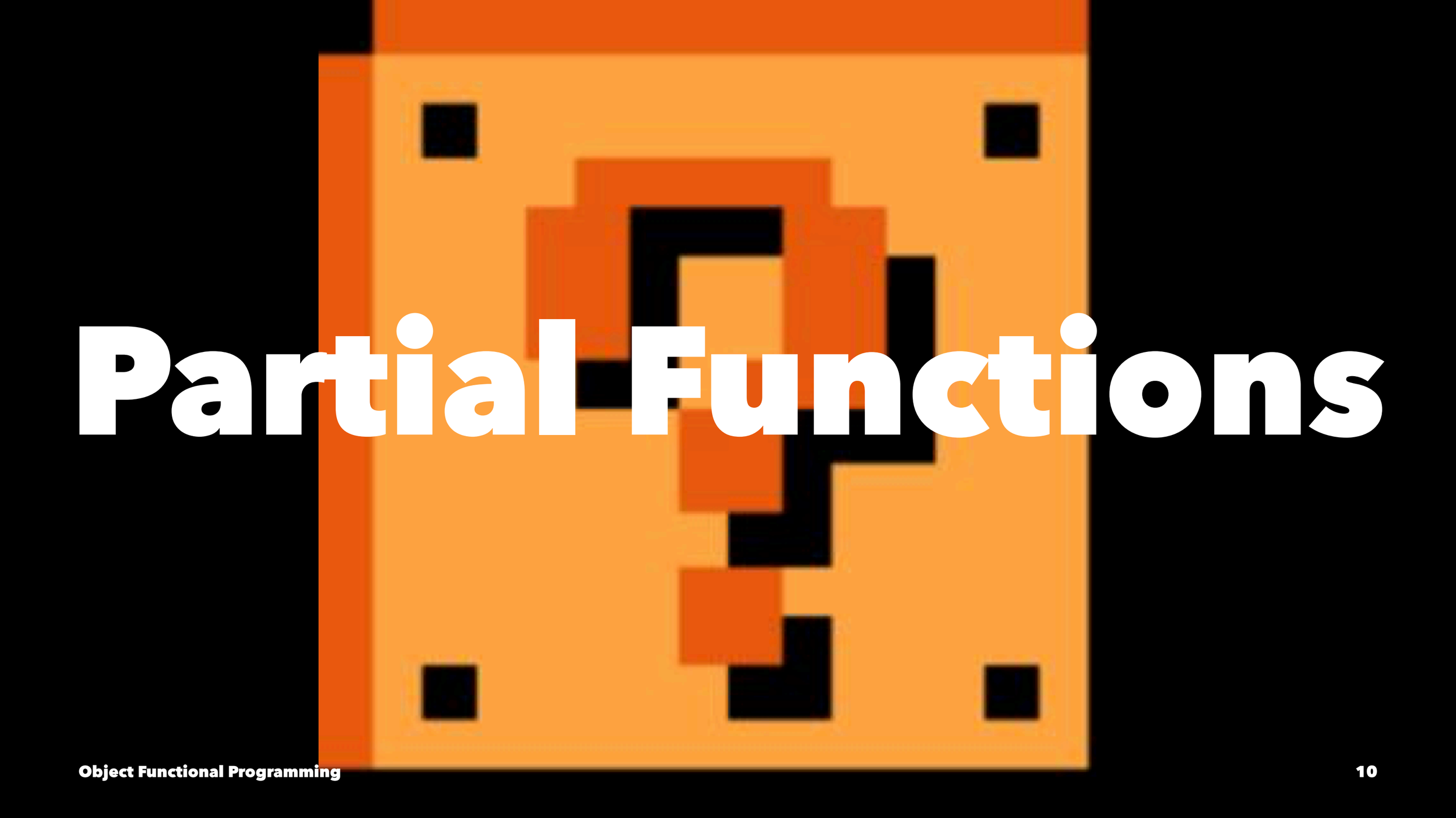
```
def factorial(n: Int): BigInt = {
  @tailrec
  def fact(n: Int, acc: BigInt): BigInt =
    if (n == 0) acc else fact(n - 1, n * acc)

  fact(n, 1)
}
```

Here we use a local tail recursive method.
Our exposed `factorial` method has the *expected* signature.

Exercise #1

Tail Recursive Operations

- Add a `totalPrice` method to the `Itinerary` companion object, which will determine the total cost of a given instance of `Itinerary`.
- Return the total cost of the given `Itinerary`
- **HINT**: We will use Squants `Money` instances for price (they should already be in `Flight`)
- **NOTE**: You must make your method tail recursive.
- Verify the provided tests pass.



Partial Functions

Partial Functions

In mathematics, a partial function does not need to be defined on its whole domain, e.g. within real numbers, $\sqrt{x}$ is only defined on non-negative x.

Scala has a `PartialFunction` trait that extends `Function1` with an `isDefinedAt` method so that users can test if the function is defined before calling it.

```
trait PartialFunction[A, B] extends A => B {
  def isDefinedAt(a: A): Boolean
  def apply(a: A) : B
}
```

Defining a Partial Function Directly

```
val sqrt = new PartialFunction[Double, Double] {
  def isDefinedAt(a: Double) = a >= 0
  def apply(a: Double) = scala.math.sqrt(a)
}

List(25.0, -4.0, 9.0).collect(sqrt)
// res25: List[Double] = List(5.0, 3.0)
```

Collect is similar to `map` but skips elements where the partial function is not defined.

Defining Partial Function Literals

A partial function literal is given with a block of *case* alternatives:

```
val sqrt : PartialFunction[Double, Double] = {
  case a if a >= 0 => scala.math.sqrt(a)
}
```

Such implementations check the match conditions in both `isDefinedAt` and `apply`:

```
sqrt(-1)
// scala.MatchError: -1.0 (of class java.lang.Double)
```

Partial Function Literals vs Function Literals

Thanks to Scala's pattern matching, this syntax may sometimes be more readable than the usual function literal:

```
val gates = Seq(("A", 1), ("D", 8), ("C", 2))
gates.map(pair => pair._1 + pair._2)
gates.map{case (letter, index) => letter + index}
// res0: Seq[String] = List(A1, D8, C2)
```

- **NOTE**: Be careful about non-exhaustive matches.
- **Quiz**: Is this example's match exhaustive?

Collections as Partial Functions

```
// Map[K, V] extends PartialFunction[K, V]
Map("a" -> 1).isDefinedAt("a")
// res0: Boolean = true

// Seq[A] extends PartialFunction[Int, A]
Seq("a", "b", "c").isDefinedAt(4)
// res1: Boolean = false
```

- **Quiz**: What about `Set` ?

Partial Function Alternatives

`PartialFunction` provides the `orElse` method to compose a fallback with another `PartialFunction`:

```
(Map(1 -> "a") orElse Map(2 -> "b")).isDefinedAt(2)
// res0: Boolean = true
```

- The `orElse` alternative will be run only if there are no matches on the primary `PartialFunction`.

Important collection methods: `exists`

The `exists` method tests whether a given function returns **true** on **some** element of a collection:

```
trait Traversable[A] {
  def exists(p: A => Boolean): Boolean
  // ...
}

Seq(1, 2, 3, 4).exists(_ % 2 == 0)
// res0: Boolean = true
```

- **NOTE**: `exists` "succeeds fast", i.e. as soon as it finds a match, it returns `true`:

Important collection methods: `forall`

The `forall` method tests whether a given function returns **true** on **all** elements of a collection:

```
Seq(1, 2, 3, 4).forall(_ < 4)
// res0: Boolean = false

Seq().forall(_ == 5)
// res0: Boolean = true
```

Important collection methods: `sliding`

The `sliding` function groups the elements of a collection into fixed sized blocks by passing a “sliding window” over them:

```
trait Iterable[A] {
  def sliding(size: Int): Iterator[Iterable[A]]
  // ...
}
```

Important collection methods: `sliding`

By Example

NOTE: `sliding` returns an `Iterator`, and therefore is lazy:

```
val groups = 1 to 4 sliding 2
// groups: Iterator[scala.collection.immutable.IndexedSeq[Int]] =
//   non-empty iterator

groups foreach println
// Vector(1, 2)
// Vector(2, 3)
// Vector(3, 4)
```

Exercise #2

Use `forall`, `sliding`, and `PartialFunction`

- We can use `forall`, `sliding`, and a `PartialFunction` to implement code with similar behavior to our Tail Recursive method.
- Add a new `isScheduleIncreasing` method to the `Itinerary` companion object, which checks whether or not a given `Itinerary` is increasing in time (i.e. you won't miss a flight segment)
- Verify the provided tests pass.
- **Quiz**: What are the pros & cons of this implementation versus tail recursion?

Currying

Currying Methods

Currying is the transformation of a function taking multiple arguments into a *chain* of functions–each of which take only one:

```
def plainAdd(x: Int, y: Int) = x + y
def spicyAdd(x: Int)(y: Int) = x + y
```

In Scala a **curried method** has more than one parameter list; each of these can have one or more parameters.

Curried Methods

To invoke a curried method, we must provide values for *all* argument lists:

```
spicyAdd(1)(2)
// res0: Int = 3
```

Doing so will then complete – and therefore execute – the function.

Partially Applied Functions

What would happen if we *didn't* provide all of the argument lists?

```
spicyAdd(1)
// <console>:9: error: missing arguments for method spicyAdd;
// follow this method with '_' if you want to treat
// it as a partially applied function
```

This isn't quite what we are aiming for, as this code doesn't compile.[¶]

[¶] However, the compiler gives us a hint about how to fix it.

Partially Applied Functions

If we replace one or more of a function's argument lists with the underscore (_), we can convert a function into a **partially applied function**:

```
val addOne = spicyAdd(1) _
// addOne: Int => Int = <function1>

addOne(2)
// res0: Int = 3

spicyAdd _
// res1: Int => (Int => Int) = <function1>
```

Applying only *some* of the argument lists will return a new function which expects only the missing arguments.

Partially Applied Functions

Lifting Methods into Functions

If the signatures “match” a method with missing argument lists, it will be “lifted” into a function automatically:

```
1 to 3 map spicyAdd(1)
// res0: IndexedSeq[Int] = Vector(2, 3, 4)

val addOne: Int => Int = spicyAdd(1)
// addOne: Int => Int = <function1>
```

A versatile collection method: `foldLeft`

The `foldLeft` method traverses a collection to produce some value.

It takes an initial value `b` and binary function `f`.

```
trait Traversable[A] {
  def foldLeft[B](b: B)(f: (B, A) => B): B
  // ...
}
```

The function is applied to the initial value `b` and the first element, and then again on the result and the next element, and so one until the whole collection is traversed:

```
List(1, 2, 3).foldLeft(b)(f) === f(f(f(b, 1), 2), 3)
```

A similar fold: `foldRight`

The `foldRight` method is similar to `foldLeft` but traverses the collection in the reverse order.

The order of the parameters of the binary function `f` are also reversed to reflect this:

```
trait Traversable[A] {
  def foldLeft[B](b: B)(f: (B, A) => B): B
  def foldRight[B](b: B)(f: (A, B) => B): B
  // ...
}

List(1, 2, 3).foldLeft(b)(f) === f(f(f(b, 1), 2), 3)
List(1, 2, 3).foldRight(b)(f) === f(1, f(2, f(3, b)))
```

Folds

By Example

Quiz: What's the result of the following expression?

```
(1 to 3).foldLeft(0)(_ + _)
```

Quiz: What about if we go from the right, instead of the left?

```
(1 to 3).foldRight(0)(_ + _)
```

- In both cases, the answer is 6.

Exercise #3: Calculate Travel Time

Using `fold`

- Add a `totalFlightTime` method to the `Itinerary` companion object, which will determine the total in flight time of a `Seq[Flight]`.
- You need to return an instance of JodaTime `Period`.
- **HINT**: There are helper methods on `Flight` and `Schedule` for determining the duration, and subtracting two instances from one another for the time between.
- **NOTE**: Make sure you use one of the flavors of `fold` to perform the calculation.
- **NOTE**: We only care about total time in the air, *not* including layover times.

Exercise #4: Plan Flight Itineraries

- This is a more complex exercise; you'll find the `man  e` command useful to keep track of the instructions.
- Our goal is to propose one or more `Itinerary` based on a `Seq[Flight]`, taking into account a 'minimum connection time', i.e. will I have time to catch my connecting flight?
- First, you'll want to create a new `FlightPlanner` class under `training.scala.air_scala.scheduling`, with a class parameter `availableFlights` of `Seq[Flight]`.
- Define a new `proposeItineraries` in `FlightPlanner`, taking 4 arguments:
  - `from` and `to`: Instances of `AirportCode`, which must not be equal
  - In a 2nd parameter list, `minConnectionTime`: A JodaTime `Duration`, defaulting to 90 minutes.
  - In a 3rd parameter list, `maxConnections`: The maximum number of stops between `from` and `to`, defaulting to 2
- **NOTE**: Please use `zip` instead of `sliding`.
- Verify the provided tests pass.

Pattern Matching

Extractors Objects

Review: Tuple and Constructor Patterns

```
val (a, b) = (1, 2)
// a: Int = 1
// b: Int = 2

val Some(v) = Some(5)
// v: Int = 5
```

Case classes support pattern matching by automatically defining an `unapply` method in their companion object.

Custom Extractor Object

```
class Name(val first: String, val last: String)

object MyExtractor {
  def unapply(name: Name) : Option[(String, String)] =
    Some((name.first, name.last))
}

val MyExtractor(a, b) = new Name("John", "Doe")
a: String = John
b: String = Doe
```

Variable Number of Patterns with `unapplySeq`

```
class NonEmptyList[A](val head: A, val tail: List[A])

object NonEmptyList {
  def apply[A](head: A, tail: A*) = new NonEmptyList(head, tail.toList)
  def unapplySeq[A](l: NonEmptyList[A]) : Option[Seq[A]] =
    Some(l.head :: l.tail)
}

NonEmptyList(1, 2, 3) match {
  case NonEmptyList(_) => "1 element"
  case NonEmptyList(_, _) => "2 elements"
  case NonEmptyList(_, _, _, _*) => "3 or more elements"
}
// res0: String = 3 or more elements
```

Regular Expressions

```
val gate = "([A-Z])(\\d)".r
gate: scala.util.matching.Regex = ([A-Z])(\d)

gate.unapplySeq("A1")
// res0: Option[List[String]] = Some(List(A, 1))

val gate(letter, number) = "A1"
letter: String = A
number: String = 1
```

Exercise #5: Extractors Objects

Using `unapply`

While `FlightNumber` is a case class, it still requires us to construct it with independent parameters - `airlineCode: String` and `flightNumber: Int`. Since it's a case class, we've already been provided an `unapply` extractor. It would be nice, however, if we could also extract from a bare `String`.

- Create a Companion Object for `FlightNumber`, and define a new extractor method which, given a `String`, attempts to extract an `airlineCode` and `flightNumber`.
  - **HINT**: You can use the following RegEx to extract a flight number cleanly:

```
val FlightNumRE = "([A-Z]{1,4})(\\d{1,4})".r
```

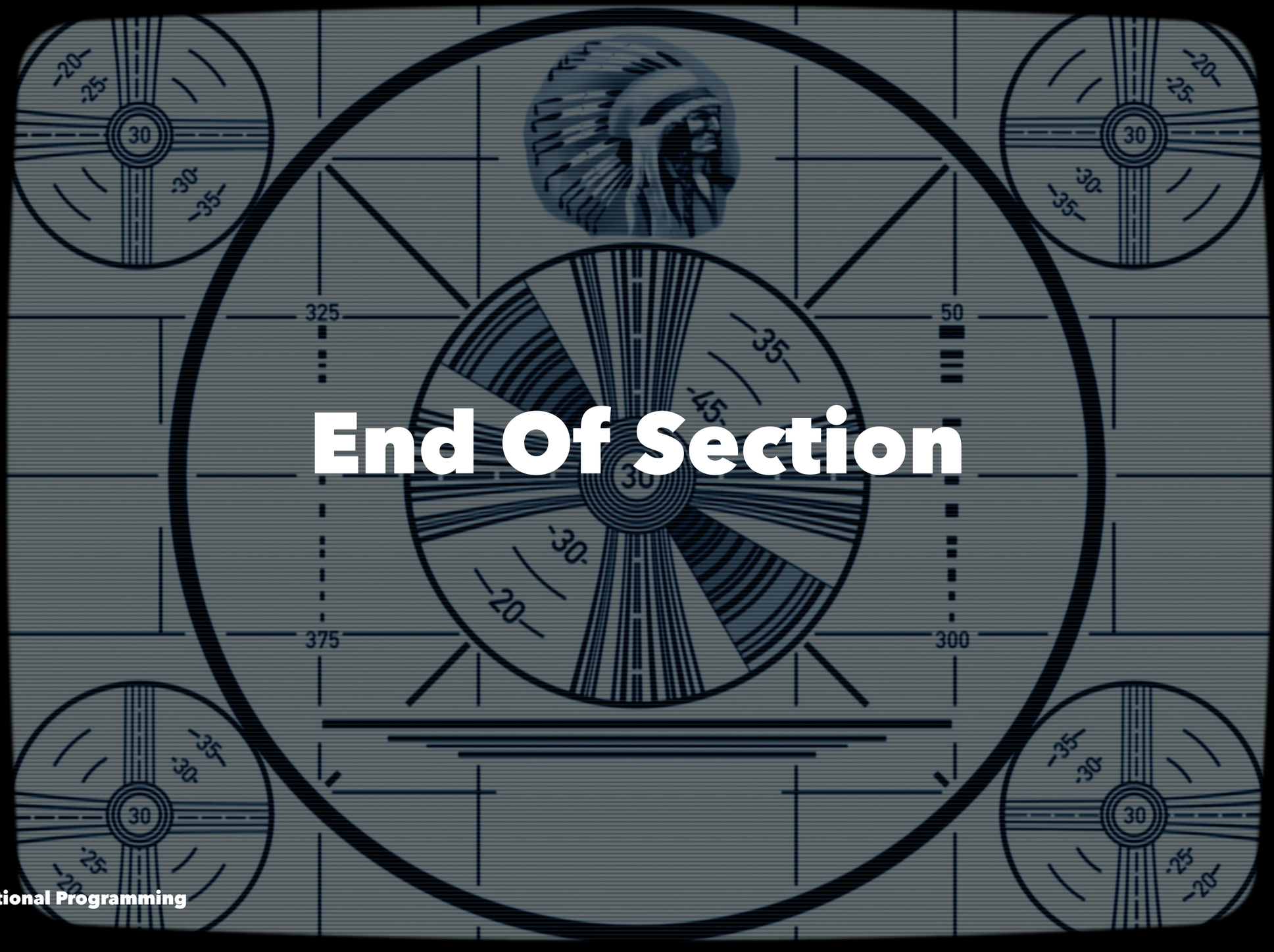
- In the Test Console - `sbt > test:console` - play with some pattern matches against instances of `FlightNumber`.
- Verify the provided tests pass.

Exercise #6: Custom Sequence Extractors

Using `unapplySeq`

Given that an `Itinerary` is made up of a `Seq[Flight]`, this might be an ideal place for `unapplySeq`.

- Define an `unapplySeq` method in `Itinerarys` companion object, which allows us to extract the Sequence of `Flights` from an `Itinerary`.
- In the Test Console – `sbt > test:console` – play with some pattern matches against the `Itinerary.unapplySeq`.
- Verify that the provided tests pass.


End Of Section